TAMPERE UNIVERSITY OF TECHNOLOGY

Markus Moisio

Compiler Implementation For a New Embedded Processor Architecture

Master of Science Thesis

Tarkastajat: Prof. Jari Nurmi
TkT Fabio Garzia
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunnan
kokouksessa 03.06.2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Sähkötekniikan koulutusohjelma
Markus Moisio: Compiler Implementation for a New Embedded Processor
Diplomityö, 46 sivua, 3 liitesivua
Kesäkuu 2010
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Jari Nurmi, TkT Fabio Garzia
Avainsanat: Kääntäjät, prosessorit, porttaus, GCC

Transistorien integrointitiheys on jatkanut kasvuaan jo vuosikymmeniä, eikä loppua ole näköpiirissä. Suunnittelijoiden tuottavuus ei valitettavasti ole pysynyt tämän kehityksen perässä, ja tämä on kasvattanut piirien suunniteluaikoja liikaa.

Ongelman ratkaisemiseksi on siirrytty käyttämään enemmissä määrin uudelleenkäytettäviä lohkoja, joita voidaan yhdistellä monin eri tavoin, ja koostamaan piirit näistä ns. IP-komponenteista. Yksi tällainen IP-komponentti on Tampereen teknillisellä yliopistolla kehitetty RISC-prosessori, COFFEE.

Jotta prosessori olisi hyödyllinen, se kuitenkin tarvitsee monia eri ohjelmointityökaluja kuten kääntäjän, assemblerin ja linkkerin. Tässä työssä on kuvattu kääntäjän toteutus kehitetylle COFFEE RISC-prosessorille.

Aluksi tutkimme mahdollisia toteutustapoja, joita oli kolme: tehdä kokonaan uusi kääntäjä, käyttää kaupallisia kääntäjänkehitystyökaluja, tai muokata olemassaolevia vapaan-lähdekoodin kääntäjiä. Kokonaan uuden kääntäjän kehitys ei ollut resurssien rajallisuuden vuoksi kovinkaan hyvä vaihtoehto. Kaupallisista kääntäjänkehitystyökaluista tutkimme CoWaRen LisaTek-työkalua, joka on tarkoitettu helpottamaan prosessorien ja niiden ohjelmointityökalujen kehitystä. LisaTek-työkalulla voi generoida kääntäjän graafisen käyttöliittymän avulla, mutta kaupallisena ohjelmana se asetti rajoituksia generoidun kääntäjän antamisessa vapaaseen käyttöön. Lisäksi työkalun kehitys oli vielä kesken työn kriittisimmässä vaiheessa. Tämä jätti ainoaksi ja parhaimmaksi vaihtoehdoksi valita olemassa oleva kääntäjä, ja muokata sen lähdekoodia toimimaan COFFEE RISC-käskykannalla.

Vapaasti levitettäviä uudelleenportattavia kääntäjiä löytyi kaksi: LCC ja GCC. LCC on opetuskäyttöön kehitetty uudelleenportattava C-kääntäjä. Se oli yksinkertaisuutensa vuoksi hyvin houkutteleva vaihtoehdo, mutta GCC teknologisesti kehittyneempänä ja suositumpana vei voiton. Lisäksi GCC:n myötä saataisiin myös paljon jo kehitettyjä C-ohjelmointikielellä tehtyjä ohjelmistoja käyttöön, kunhan GCC saataisiin portattua COFFEE:lle.

GCC on alusta asti suunniteltu uudelleenkäyttöä varten, ja sen rajapinnat mahdollistavat uusien kielien ja uusien käskykantojen lisäyksen. GCC:n toiminta on jaettu kolmeen itsenäiseen osaan. Yksi osaa vastaa lähdekoodin lukemisesta ja

analysoinista, keskimmäinen osa optimoinneista ja loppuosa vastaa konekielisen koodin tuottamisesta. Jotta GCC saataisiin toimimaan COFFEE RISC:n käskykannalla, sille tarvitsi kehittää uusi loppuosa.

Porttaaminen tapahtuu siten, että GCC:lle kerrotaan, miten tarvittavat aritmeettiset ja datansiirto-operaatiot toteutetaan prosessorin käskykannalla. GCC:n taustalla on kuvitteellinen ideaaliprosessori, jolle se generoi koodia, ja tämän kuvitteelisen prosessorin operaatiot ja niiden vastaavuus oikeaan prosessoriin kuvataan ns. RTL-kielellä.

GCC:n porttausmanuaalissa kerrotaan kaikki mahdolliset kuvitteellisen prosessorin käskyt, joiden toiminta oikealla prosessorilla voidaan kuvata. Näitä kaikkia ei kuitenkaan tarvitse toteuttaa, sillä GCC osaa automaattisesti emuloida nämä operaatiot niiden kuvauksen puuttuessa, tai sille voidaan antaa konekielellä tehty toteutus, jota GCC kutsuu tarvittaessa. Ainoat tarpeelliset kuvaukset ovat yhteenlasku tavuille, tavujen siirto muistin ja prosessorin välillä, loogiset operaatiot tavuille ja ehdolliset hyppykäskyt. Kun nämä on toteutettu, GCC osaa kääntää perus C koodia uudelle prosessorille.

COFFEE RISC:lle suurin osa käskyistä voitiin toteuttaa RTL-kielellä. Ainoastaan jakolaskuun liittyvät operaatiot jouduttiin tekemään konekielellä, sillä COFFEE:lla ei ole laitteistolla toteutettua jakolaskua. GCC tarvitsi myös muutamia C:n standardikirjastojen funktioita konekielellä toteutettuna, joita olivat esim. memcopy (muistilohkojen kopiointi) ja memset (muistilohkojen alustus).

C-kieltä porttauksessa on tarvittu määrittelemään datatyyppien koot, rekisterien määrät ja tarkoitus sekä pinon ja funktiokutsujen toteutustapa. Nämä on toteutettu pääasiassa C-kielisillä makroilla, jotka voidaan tarvittaessa laajentaa normaaleiksi funktiokutsuiksi, jos niiden koko on liian iso. Nämä konventiot on dokumentoitu, jotta mahdollistettaisiin muilla ohjelmointikielillä tai konekielellä toteutettujen ohjelmien integroimisen C-kielellä toteutetun koodin kanssa.

Alkutestaukseen käytettiin kahta tunnettua signaalinkäsittelyalgoritmia, jotka olivat FIR-suodatus ja DFT-muunnos. Nämä algoritmit toteutettiin C:llä ja käännettiin COFFEE RISC:lle portatulla GCC:llä. Tarkistimme, että tekemämme kääntäjä tuotti toimivaa koodia. Lisäksi tehdyt C-koodit käännetiin vielä kahden muun prosessorin GCC-kääntäjällä ja analysoitiin koodin tehokkuutta. Porttattu COFFEE GCC-kääntäjä jäi koodin tehokkuudessa osittain jälkeen verrokkiprosessoreista (ARM7 ja Pentium 4), mutta siedettävissä määrin, ja DFT-muunnoksesta se suoriutui jopa paremmin kuin ARM7. Jälkeenjääminen tehokkuudessa kuitenkin johtui suuremmalta osin COFFEE:n käskykannasta eikä varsinaisesti portatusta kääntäjästä. COFFEE:n käskykanta on huomattavasti suppeampi kuin verrokkina käytetyillä kaupallisilla prosessoreilla.

Kun pahimmat virheet olivat karsiutuneet pois, kääntäjän toimintaa kokeiltiin

muutamilla isommilla ohjelmilla. COFFEE RISC prosessori oli syntesoitu tarvittavine oheislaitteineen Altera STRATIX:n FPGA-prototyyppialustalle. Alustalla oli myös VGA-portti, joka mahdollisti grafiikan esittämisen monitorilla. Projektin ohessa oli kehitetty 3D grafiikan tekemiseen tarvittavia algoritmeja, joita käytettiin yksinkertaisen kuution pyörittämiseen ruudulla. Toinen sovellus oli H.264 videon dekoodaus, joka toteutettiin C:llä ja käytettiin onnistuneesti dekoodaamaan lyhyt video COFFEE:lla.

Tulevaisuudessa voisimme siirtyä käyttämään uudempaa versiota GCC:stä. Tällä hetkellä käytössä on versio 3.4.4, ja siirtyminen 4.x versioon parantaisi koodin tehokkuutta ja toisi tuen OPENMP-kielelle. Lisäksi C:n standardikirjastojen ja Linuxin porttaus toisivat paljon uusia mahdollisuuksia COFFEE RISC-prosessorille.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Electrical Engineering
Markus Moisio, Compiler Implementation for a New Embedded Processor
Master of Science Thesis, 46 pages, 3 Appendix pages
June 2010
Major: Embedded Systems
Examiners: Prof. Jari Nurmi, PhD Fabio Garzia
Keywords: Compilers, Embedded Systems, Processors, GCC

The department of computer systems in Tampere University of Technology has created an embedded RISC processor, COFFEE, to be used as part of System-on-Chips (SoC). These SoCs include all the hardware a device needs in a single solicon chip. Typically a SoC is constructed from readymade Intellectual Property-blocks (IP-blocks), which are designed to be reusable. A processor is on such block.

A processor itself is of very little use. To fully exlploit the potentials of processors, they need a set of software development tools: compiler, assembler, linker, simulator etc. The purpose of this thesis was to develop a high level language compiler for the developed COFFEE RISC core.

At first, different ways of reaching this goal was briefly analyzed, and based on that, the retargetable open source Gnu Compiler Compiler Collection (GCC) was chosen to be retargeted to the COFFEE RISC core.

The process of retargeting GCC required the generation of a new back-end for it. The back-end consists of a special machine description describing the basic instructions of the processor and C code.

A new back-end for GCC was created, and the correctness and performance of the created assembly code was analyzed with basic signal processing algorithms created in C. After initial testing phase, we created some larger applications such as 3D graphich algoriths and a H.264 decoder, which were tested on COFFEE RISC core running in an Altera FPGA prototyping board.

PREFACE

This thesis has been made possible by the Tampere University of Technology and its Department of Computer Systems.

Many thanks to my supervisor Jari Nurmi and his team of Researchers. Especially Juha Kylliäinen for his hard work on the processor core and making this whole project possible. Other persons I would like to thank are Claudio Brunelli and Fabio Garzia who have forced me to learn the insights of Italian culture and their language. Also I would like to thank all the other international trainees I have gotten to know during these years. Jari Nurmi's group has a very international atmosphere and that makes it delightful working environment.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CISC	Complex Instruction Set Computer
FP	Floating Point
FPGA	Field Programmable Gate Array
GCC	Gnu Compiler Collection
GNU	Gnu's Not Unix
GPL	General Public License
HLL	High Level language
IC	integrated Circuit
IP	Intellectual Property
ISA	Instruction Set Architecture
LCC	Local C Compiler
MD	Machine Description
MMU	Memory Management Unit
OS	Operating System
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language or Register Transfer Level
SOC	System-On-Chip
VHDL	Very high speed integrated circuit Hardware Description Language

1. INTRODUCTION

Usage of processors in different applications is constantly increasing and more and more applications are implemented mainly in software instead of hardware. This trend is justified by the fact that silicon manufacturing technologies continue to advance at the Moore's law rate. That is every two years the number of transistors in integrated circuits is doubled. This law has led to a so called *design productivity gap*, which means that traditional methods of designing integrated circuits are not adequate anymore because the design productivity of an engineer is not increasing at the same speed. New methods are therefore needed to ensure that design times do not rise to unacceptable levels.

One solution to this problem is the *Intellectual Property(IP) reuse* methodology. Instead of designing systems from scratch the trend is towards designing reusable blocks, which can then be used again in different applications. These blocks can then be combined as needed in a so called *System-On-Chip* [10]. One integral part of most SOCs is a general purpose processor. Modern manufacturing technologies are able to produce processors in the GHz range so there is less demand for special purpose hardware, because these high performance processors enable us to do a larger proportion of the application in software. However to enable the powerful properties of software one needs a *high level language*(HLL) compiler.

The purpose of the HLL compiler is to raise the abstraction level of software development. In the early days computers were programmed in processor specific languages (generic term: assembly language). Software written in these languages were not portable to another processor and required detailed knowledge of the processor and its *instruction set architecture* (ISA). HLL compiler hides the underlying processor and provides a standard language for programming. This standard language is used to develop applications, which can then be built on any processor for which a similar compiler exists, without modifying the original source code.

One of the first HLLs developed is the C language [2]. It was developed in 1970s by Dennis Ritchie at the Bell Telephone laboratories. The main design goals of C were:

- Straightforward compilation with a simple compiler
- Low-level access to memory
- Very little run time support required
- Machine independent programming

These properties were powerful enough to spur the reimplementation of the Unix operating system in a language other than assembly. Also being a simple compiler to implement, C compilers were developed for many other processors and its usage as a principal programming language quickly spread. The quick adoption of C, its large application base and its powerful properties make it the most widely used language in embedded systems programming, even today.

The Department of Computer Systems in Tampere University of Technology has developed a reusable processor core targeted to SOCs. This thesis describes the development of a HLL compiler for this processor named the COFFEE RISC core, the different methods of realizing a compiler, what kind of method was chosen and the results obtained during this process.

Chapter 2 gives a brief introduction to the COFFEE RISC core and the background of this project, focusing on properties which are relevant to a HLL compiler. Then the third chapter reviews different approaches for achieving this goal and focuses on what approach was chosen. An important part of the compiler development is the definition of the processors *Application Binary Interface* (ABI), which describes the important conventions regarding data types and function calling sequences. The COFFEE ABI is the focus of chapter four. The fifth chapter describes the details about porting GCC to a new architecture and its internal architecture which enables it to be a portable HLL-compiler. The chapter tries to give a good tutorial into the porting mechanism of GCC, but most of the technical details are left out. Because The COFFEE RISC core does not have byte loads and stores, it was required to work around this limitation through other means. This mechanism is described in the sixth chapter. Chapter seven focuses on the testing of the COFFEE port of GCC. It introduces the platform used for testing the COFFEE RISC core and the other software tools necessary for programming. During the project many different applications have been developed and run on the COFFEE RISC core. A few of the applications and their results are introduced in this chapter. Finally we have the conclusions which wrap up the experiences gained during this project, and

give some information about the ideas for the future development of the COFFEE GCC compiler port.

2. THE COFFEE RISC CORE PROJECT

The goal of COFFEE RISC project was to develop a general purpose RISC core to be used as an IP-block in SOCs. This means that COFFEE should be easy to integrate and implement in many different platforms, whether they are *Field Programmable Gate Arrays*(FPGA) or *Application Specific Integrated Circuits*(ASIC). Such a processor should also have enough performance to support most applications, which are used in modern embedded systems. To accomplish these goals it was implemented in textitRegister Transfer Level (RTL) *VHDL* (Very high speed integrated circuit Hardware Description Language). In addition we decided to make it freely available for anyone, in order to increase the number of users. This idea is based on the open source principle used in software. Detailed information about the core is available in M.Sc thesis [1] by Juha Kylliäinen, and a good brief overview of the architecture and the consepts adopted can be found in [28].

A single processor core needs additional hardware and software components to be used for application development. To develop software for a processor the typical tools are the HLL compiler, assembler and linker. The HLL compiler creates assembly code for the assembler, the assembler creates the actual machine code out of assembly language text, and finally the linker is used to combine different software components together as a single program.

Additional hardware components that increase the usability of a processor include: timers, floating point co-processor, direct memory access-controller, etc.

During the years many applications have been developed for the COFFEE RISC and some of the resulting work is publicly available for anyone to use and modify in the COFFEE RISC core website http://coffee.tut.fi.

2.1 COFFEE RISC core Overview

The different technical decisions of processor design are very important from the compilers point of view. Traditionally general purpose processors were not designed to take into account the HLL compilers and their development. But as soon as software development saw the first HLL compilers, processor development needed to shift focus more on the support for HLL compilers instead of hand-coded assembly.

As COFFEE is a RISC processor it has a very clean and simple instruction set. Before HLL compilers processors were mostly CISC (Complex Instruction Set Computers, i.e., one instruction had many functions it could accomplish). When most of the programming was done with hand-coded assembly it made sense to have such instructions. However this approach has the drawback of increasing the complexity to develop a HLL-compiler for these processors. As a result compilers tended to only use a small subset of possible instructions available. Unused instructions only waste resources in every sense: they increase power consumption and area occupied on a silicon chip.

Here is a list of the main features of the COFFEE RISC core which affect the HLL-compiler in some way:

- Six-stage pipeline
- Harvard architecture
- Separate modes for user and privileged mode for operating systems
- Two register banks, 32 32-bit registers each for user and privileged user
- Up to four coprocessors can be connected to speed up different applications
- 16-bit and 32-bit instructions encodings
- Conditional execution of instructions

2.2 Processor Properties Relevant to the Compiler

All the decisions made in the design phase of a processor impact the compiler development immensely, and it is advisable to have tight integration between the hardware and software designers right from the beginning. Compiler development is the more demanding part of the process nowadays, and the task of creating an efficient compiler should not be made more difficult without properly analyzing the pros and cons of all the hardware properties and processor instructions. As COFFEE is a RISC processor at heart it is a good target for compilers in general.

2.2.1 Pipeline Data Hazards

The pipeline and its possible data hazards are one of the most important features from the compiler's perspective. Every time an instruction entering the pipeline requires a result from a earlier instruction, there is a possibility of a data hazard. A data hazard occurs when a result is not ready before the instruction that uses it. Without taking these data hazards into account (in the compiler or the hardware) you have data corruption, which can eventually crash the whole program.

In the hardware you can address these data hazards with a dedicated detection logic. When a data hazard is noticed in the instruction flow you have two choices: either stall the pipeline or forward the data inside the pipeline to a previous stage. Stalling the pipeline means that the instruction using a previous instruction's result is not allowed to proceed in the pipeline before the result is available for it to use. Forwarding means that the result is rerouted inside the processor to a previous stage so that the next instruction can use it before the previous instruction has gone through the whole pipeline. Forwarding is the best choice but this is not always possible. Then you have to stall the pipeline. This is done by forcing a no-operation instruction in the pipeline and stopping the progress of previous stages.

If the hardware is not designed to handle these situations then the compiler has to do it. In particular the compiler has to know all the possible data hazards and either insert sufficient number of nops between instructions to avoid data corruption, or change the instruction sequence without altering the meaning of the program.

2.2.2 Delay Slots

Many processors have so called delay slots. These delay slots are associated with the branch instructions of a processor. If a processor has a delay slot in a branching instruction, it means that the instruction following the branch instruction is always executed. When the branch instruction is conditional, this has to be taken into account by the compiler.

There are two ways to handle this problem: either put in a no-operation instruction after every branch instruction, or find a suitable instruction to be put there.

Fortunately GCC has an integrated method of describing the possible delay slots of a processor, and it is used in the COFFEE compiler port also. The GCC can be notified that an instruction has a delay slot, so it can automatically find an instruction to be placed in the delay slot. If the compiler cannot find a suitable instruction it places a no-operation instruction in the delay slot.

2.2.3 Registers

The number of registers on a processor has significant effects on the performance of the compiled code. Having a large amount of registers helps to reduce memory accesses, which is the usual bottleneck in modern systems. With a large amount of registers the compiler can keep variables and intermediate results inside the registers and reduce stack accesses, which in turn reduces memory accesses and improves performance.

The COFFEE has large register banks for both the regular user and the superuser from the compilers point of view, and helps the allocation of registers to different purposes and reduces the usage of stack between function calls.

2.2.4 Separate Modes for Superuser and Regular User

To support modern *Operating Systems*(OS) COFFEE has two different sets of registers and operating modes: one for superusers and one for regular users. The regular user has limited access to memory and certain instructions are forbidden, which prevents applications from conflicting with each other's data or code accesses. Violations result in an exception and the execution is given to the superuser to find the problem and possibly fix it, or terminate the offending application.

Typically C code is written to be executed in regular user mode, and programming in superuser mode is very rare and requires special procedures from the programmers point of view. The common way of switching modes of a processor from regular user to superuser goes through a special instruction. The mode switching instruction is not needed to create executable programs from C, so the switching of modes in C has to be done either by embedding assembly code in to C, or to call a function written in assembly from C.

Only the OS is typically run in superuser mode and all applications are run in regular user mode to have the benefit of memory protection and multitasking. Of course, on systems without OSs the applications are executed in superuser mode to have full hardware access to peripherals, but the default is to assume that the applications are run in control of an operating system.

2.2.5 Predication

COFFEE supports predicated execution of instructions, which means that most of the instructions can be coded with a condition to check whether to execute it or discard it. This speeds up the execution of small loops and conditional code because branching is reduced. Usually a conditional block is skipped by jumping over it if the condition is false, but conditional execution removes this jump and all the instructions are passed through the pipeline without executing them. Also

looping over a code block benefits from this. However, this is only true for a small conditional/loop blocks. While eliminating jumps and the resulting pipeline flushes are beneficial, feeding useless instructions increases the global latency and power consumption. By experience we can say that only a very small conditional/loop block benefits from conditional execution of instructions, while a loop with more than 4 instructions is faster to execute without conditional instructions. As a result the COFFEE compiler port does not support this feature.

2.2.6 Instruction Coding Size

During the early stages of designing the COFFEE RISC core, memory on embedded systems was limited, and it was beneficial to have a 16-bit encoding of instructions to save precious memory. But with the increase of available memory on-chip the usefulness of this feature has reduced and it is not used much anymore. Also the limitations introduced with shorter encoding of instructions results in a reduced performance.

2.2.7 Floating Point Coprocessor

As part of the COFFEE RISC project a floating point coprocessor (MILK) was developed to boost the performance of some applications, for example 3D graphics.

MILK provides many arithmetic operations for the IEEE standard 754-1985 for single precision floating point numbers. The COFFEE RISC core supports the MILK coprocessor. Supported instructions and their explanation is in appendix I. Some of the instructions of MILK are not supported by the compiler because of their marginal usage.

Later, when larger FPGAs were introduced we decided to integrate the floating point coprocessor into the COFFEE core. This gave us better performance, because the data transfers between the COFFEE core and the coprocessor were left out completely. This version of COFFEE RISC core was named CAPPUCCINO.

3. INTRODUCTION TO C COMPILERS AND GCC

The first C compiler was developed in the Bell Telephone Laboratories in the early 1970s [12]. Its roots are in the B and BCPL programming languages and the DEC PDP-11 computer. The B language was not suitable for developing operating systems so it went under many improvements and modifications. By 1973 it was so different that the resulting language was renamed as C. After that it was retargeted to other machines and the UNIX OS was written with it. But it was not until 1978 when the first book about C language was published [13].

During the 1980s the popularity of C started to grow rapidly and it was available to almost all processors and OSs. A language this popular was necessary to be standardized so in 1982 the standardization process started. This resulted in the ANSI standardization in 1989 [14].

3.1 Compilation Flow

Producing an executable program from the C language requires many steps. This flow is presented in figure 1.

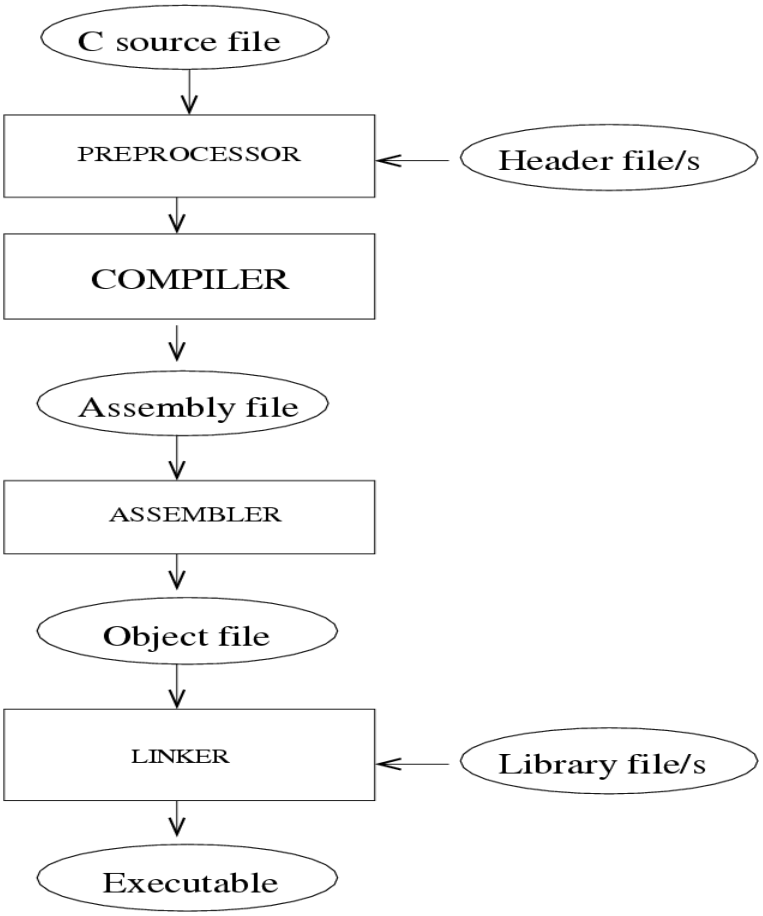

Figure 1: C language compilation flow

The first step is the preprocessing phase. The job of the preprocessor is to strip out comments and replace # *include* directives with the contents of the corresponding source file. Also macros are expanded by the preprocessor. Preprocessed source file is then given to the actual compiler, which produces assembly code from it. Then the assembler generates object file from the assembly. Object file is actual machine code but can contain external function calls or variables, which are defined in other source files. In the final phase the linker is used to combine these different object files together in a single executable file.

3.2 Different Implementation Strategies

There are three different approaches to develop a C compiler today. First one is the traditioonal method of designing and implementing it from scratch. Compared to

other modern languages C language is a relatively simple language to implement. In fact all the basic language constructs have a corresponding operation in assembly language, but the porting requires still a lot of work. Depending on how much optimization routines you want to have, the actual source code can become as large as 100k lines. Also as processors have become more complex, compiler development becomes more demanding too.

Another approach is to use an existing compiler. There are two distinct parts in compilers: the source language scanner&parser and the assembly generating part, typically called front-end and back-end. The scanner is responsible for reading the source language text and recognizing the language constructs (reserved words, variables, arithmetic statements) and creating so called tokens for the parser. Then the parser analyzes these tokens and checks whether the source code is semantically correct. Finally the statements are stored in a tree-like data structure. This data structure then goes through many optimization procedures before it is passed on to the assembly generator.

The front-end in a standards compliant compiler is not dependent on the target architecture and therefore it is possible to reduce significantly the design time if you can use an existing compiler. To do this access to the source code is needed and the compiler should be designed with retargetability in mind. This is where the emerging of *open source software*(OSS) has its advantages. There are also commercial software which give acces to the source code and permission to modify them. However, they have strict limitations on the distribution of the modified source code and they are not free for use (as in speech).

There were two retargetable compilers to choose from: The *GNU C Compiler* (GCC) and the *Local C Compiler* (LCC). The LCC was developed for educational purposes and it is well-documented in the book "A Retargetable C Compiler: Design and Implementation" by Chris Fraser and David Hanson. It is also available free of charge, but it is not open source, so sticking to the open source philosophy we chose the GCC for our purposes.

In the last years also commercial software tools offer retargetable compilers. As SOCs and the usage of processors in applications is increasing there is a demand for tools in designing application-specific processors and software development tools for them. One of the first ones in this area is the CoWare LISATek [3] software suite. With LISATek you can easily design different processor architectures and analyze the performance of different instruction sets. The tool provides a simulator and software tools based on a description resembling a programming language. However being commercial software it was not suitable in the end for our open source philosophy. Also the tools were still in development stage during the critical time of this project.

3.3 Benefits of Open Source Software and GCC

One of the most famous and successful software projects in recent years has been the GNU Linux [4] operating system. It was originally started by Linux Torvalds as a hobby but due to the thousands of volunteers and enthusiasts it has grown as a serious competitor in the operating system market. The Linux *Operating System* (OS) is nowadays even backed up by such industry giants as IBM and Nokia. The GNU addition in the name is because the tools provided by the *Free Software Foundation's* (FSF) GNU project are a very integral part in the development of the commonly known Linux OS. This OS and the tools are completely distributed as open source without any charge. One is free to use them as they wish provided that he/she agrees to provide modifications or improvements in the software to everyone else under the same *Gnu General Public License* [5].

Part of these tools is the GCC; a set of compilers mostly used under UNIX like OSes. Originally it started as a C compiler only but after 20 years of development it has grown to support also C++, Fortran, Objective-C, Java and ada. During these years hundreds of motivated and talented people have been contributing to this project and this has resulted in a very diverse and powerful HLL compiler suite. Because of the open-source nature it is able to produce highly optimized code comparable to commercial compilers. It is true that highly specialized commercial compilers (not intended to be retargetable) have some advantage in overall code quality. However, as a free, retargetable compiler it has no competitor. GCC has been ported to almost 100 different processors and several different OSes. Almost every new processor arriving to market will, at some point, have a port of GCC (and usually that is the only compiler available).It is also able to cross-compile (meaning that the target architecture is not the same as the development architecture).

The biggest advantage in our approach is that when you have the whole software development toolchain ported from these GNU tools (GCC, GAS, linker) you have access to a very large set of other applications, which require in the ideal case only a recompilation for your new architecture. The only applications that require some porting efforts are the Linux kernel and the standard C library but they have only a small portion dependent on the target architecture.

4. THE COFFEE ABI

This chapter describes the COFFEE RISC *Application Binary Interface* (ABI) used by the COFFEE GCC compiler. The purpose of the ABI is to document all the important conventions that the compiler uses to call functions, store data and how the stack is organized. Using the same conventions makes the interaction of programs written in different languages possible. It is also usable when debugging applications with aid of a special debugger, such as the GNU GDB.

4.1 Basic Data Types

Table 1 shows how the scalar types defined in the ANSI C standard are mapped on the COFFEE compiler port. The default signedness of the type (signed/unsigned) is implicated by enclosing it in parenthesis, they can be omitted from actual C code.

C type	Size(bytes)	Alignment(bytes)	COFFEE type
(unsigned) char	1	1	Unsigned byte
signed char	1	1	Signed byte
Unsigned short	2	2	Unsigned halfword
(singed) int/long, enum	4	4	Singed word
Unsigned int/long	4	4	Unsigned word
(singed) long long	8	8	Signed double word
unsigned long long	8	8	Unsigned double word
Pointer	4	4	Unsigned word
float	4	4	Single precisions float
double	8	8	Double precision float

Table 1: Size and alignment of basic C data types

The COFFEE type sizes are 8, 16 and 32 bits for byte, halfword and word respectively. Floating point numbers use the IEEE-754 standard for the float and double C data types and they are 32 bits and 64 bits in size. The COFFEE has hardware support only for the single precision floats and the double precision is supported by emulation.

Alignment shows on what memory address the respective data type should be aligned to. Each data type is to be stored on an address that is divisible by their

alignment number, i.e., every int type has to start on an address that is divisible by four.

4.2 Structures and Unions

Memory for structures and unions is allocated in the order they appear in the declaration and the structure or union starts at the word aligned address. Padding may be needed to align the next member of the structure or union to its required alignment.

For example, the following struct:

```
struct
{
  int number;
  char letter;
  float single;
  double double;
}
```

would be allocated into memory as follows:

address	1 byte	2 byte	3 byte	4 byte
0x100	number	number	number	number
0x104	letter	padding	padding	padding
0x108	single	single	single	single
0x10c	padding	padding	padding	padding
0x110	double	double	double	double
0x114	double	double	double	double

and the following struct:

```
struct
{
  short s;
  int i;
  char c1;
  char c2;
}
```

would be allocated in the following layout:

address	1 byte	2 byte	3 byte	4 byte
0x100	s	s	padding	padding
0x104	i	i	i	i
0x108	c1	c2	padding	padding

4.3 Function Calls

This section describes the conventions used by the COFFEE compiler port when calling functions in C. It covers register allocation, stack layout and the way it uses to pass parameters and return values from functions.

4.3.1 Register Allocation

The COFFEE RISC has two register banks with 32 registers each. The COFFEE compiler port only use the regular users register bank and assumes it is in use by default. The compiler does not know what register bank is in use if it is changed, e.g by the use of embedded assembly code, and it is left to the user to keep track that the right bank is in use. The register allocation scheme of COFFEE compiler is shown in table 2.

Register	Usage
R0	Incoming argument and return value
R1-R4	Incoming arguments
R5-R14	Temporary values, saved across function calls
R15-R26	Temporary values, not saved across function calls
R27	Stack pointer
R28	Frame pointer
R29-R30	Reserved for superuser
R31	Link register

Table 2: Register allocoation scheme of COFFEE compiler

4.3.2 Function Calling Sequence

Before making a function call the COFFEE compiler port places the outgoing arguments in registers R0-R4 so that the first argument in the function definition is placed in R0, then it calls the function. If there are more then 5 outgoing arguments, the rest are placed on the stack in the calling functions stack frame, which has reserved place for them during the entry of the calling function.

All the arguments are extended to 32 bits. If the argument is bigger than 32 bits, it has two register slots assigned to it in the big endian fashion. Structures and unions are passed as pointers only.

On the entry section of a function the link register and the frame pointer are saved on the stack if the function makes other function calls. Also all the temporary registers, which need to be saved across function calls used by the function are saved on the stack. If the function has a variable number of arguments then it has to store the register arguments first to the stack frame below the regular incoming arguments so that all the arguments are placed on the stack and can be accessed with the aid of the frame pointer.

On the exit of a function the possible return value is placed in the R0 register. If the return value is a structure or a union, the calling function has assigned space for it in its own stack frame and has passed the location in register R0 as an invisible first argument. The called function then copies the struct to this address. Saved temporary values are stored in their respective register slots, and the stack pointer and frame pointer are recovered from the stack, and then returns to the calling function.

4.3.3 Stack Frame Layout

Each function allocates a frame from the run-time stack. The stack is a full descending stack so it grows from high addresses to low addresses. The stack pointer points to the end of the last allocated stack, and the frame pointer points to the end of the previous functions stack frame. The incoming function arguments are then found with the aid of the frame pointer, if there are more than five or the function has a variable number of arguments, and they have a positive offset from the frame pointer. The place for outgoing arguments is reserved based on the function that has the biggest amount of arguments. If the return value from a function is structure or a union then the place for this is located in the area of stack reserved for local variables and temporaries.

Table 3 shows the details of the COFFEE compiler stack frame.

Position/size	Contents	notices
FP + N*pretend arguments	Incoming arguments excl. the first five	If needed
FP + 0	Possible pretend arguments	For variable argument functions
FP - 4	Return address	If not a leaf function
FP - 8	Saved previous frame pointer	
N*4	Place for callee's saved registers	If needed
	Place for local temporaries and variables	If needed
SP + 0	Place for outgoing arguments above the first five	If not a leaf function

Table 3: The stack frame organization of the COFFEE compiler

A function that does not call other functions (a leaf function) does not have to allocate a stack frame if it does not need to save any registers to the stack, or reserve space for its own variables.

5. PORTING PROCESS

GCC is a very complex and large program, and it is divided into three distinctive sections. This is because GCC was designed to be portable both for different source languages and for different instruction sets. For these reasons GCC has a front-end, a target machine independent section and a back-end.

The Front-end is responsible for reading the source language, it performs semantic analysis and creates a parse tree out of the program written in the source language. This tree is then converted to a language independent format (another tree structure) used inside the compiler. This way the compiler can use the same optimization algorithms on all the source languages that have been integrated to GCC. Moreover, the addition of new optimization algorithms benefits all the languages at once without any modifications. Then the compiler converts the language independent tree to a list of instructions (called *insns* in short). The *insns* describe in an algebraic form what the instruction does and the format is called the Register Transfer language (RTL). These patterns are part of the back-end of the processor, and they are also used to output the assembly instructions, which accomplishes the same task in the processors assembly language.

To port GCC to a new processor it is needed to write a new back-end for it (called the machine description). The machine description consists of a special ".md" file which describes all the useful instructions of the processor in a RTL language format and handles the conversion of RTL to actual assembly instructions. Also a C header file is needed for macro definitions. If the macro is very large, then it is recommended to move those macros to a ".c" file and make functions out of them for the purpose of readability.

The macro definitions in the header file describe some general properties of the processor,e.g. the number of registers, data type sizes, addressing schemes, and so on. A description of the possible macros can be found in the GCC internals manual [11].

It is best to choose a similar processor description from the GCC source tree and use that as your starting point in developing a new port. With the COFFEE RISC core we had many choices for the starting point, and during the development process we did not use just one but many, for example ARM, Picochip, MIPS and OpenRISC.

5.1 Basic Instructions

There is a hard-coded list of basic instructions inside the compiler, which are used in the RTL *insn* list creation process, and they are described in the standard pattern names chapter [7] of the GCC porting guide. To develop a working compiler, only a certain subset of these names are required to be implemented, so most of them can be left out of the machine description, but the more you have the better in terms of performance because the compiler automatically synthesizes the required operations from the ones you have given, or expects to have assembly routine for it if it needs this type of instruction. So unless there is a specific reason not to use the processor instruction in the machine description, it is not advisable to leave it out.

The standard pattern names described in the manual have a stub for the machine mode of the instruction. This mode is in *m/n* at the end of the name, and it is to be replaced with the respective mode of the instruction described(*qi, hi, si, di, sf, df* etc). Defining a pattern name with the size *qi* does not prevent the definition of other sizes, i.e., if the processor has an addition instruction for bytes and words the machine description should define both of them, even though the compiler can use the byte addition to implement addition for larger data types. The actual bit-sizes of these modes are described in the C header file like this:

```
#define BITS\_PER\_WORD 32
#define BITS\_PER\_UNIT 8
```

The definition BITS_PER_UNIT is the size of the smallest addressable data (usually a byte) and BITS_PER_WORD is the size of the internal register of the processor. In theory these could be set to whatever size is proper, but according to the manual the compiler internally expects the byte size to be 8, and setting it to a different value is not recommended.

Regarding the *mode* in the standard pattern names, the byte is *qi* and the word is *si*. All others have default definitions calculated from these values, but they can be set to any other value as long the sizes are meaningful (i.e., *di* (double integer) is not smaller then *si* (single integer)). All the possible data types and their descriptions are found in the GCC internals manual [16].

5.2 Machine Description

The most important part of back-end is the ".md" file. The definitions (*define_ insn, define_ expand, define_ split, define_ peephole*) in this file are used to convert the internal parse tree to a machine independent RTL format and do some necessary modifications so that the RTL conforms better to the processor in question. After all the modifications and optimizations, the final phase consists of the matching of

the created *insns* and producing assembly language out of them. Other important files are the macro definitions in the C header file and possible helper functions in the C source file. Also others can exist but they are not used in the actual compiler generation.

The different type of patterns in the ".md" file are used in different phases of compilation. Their relation to the compilation process is shown in figure 2.

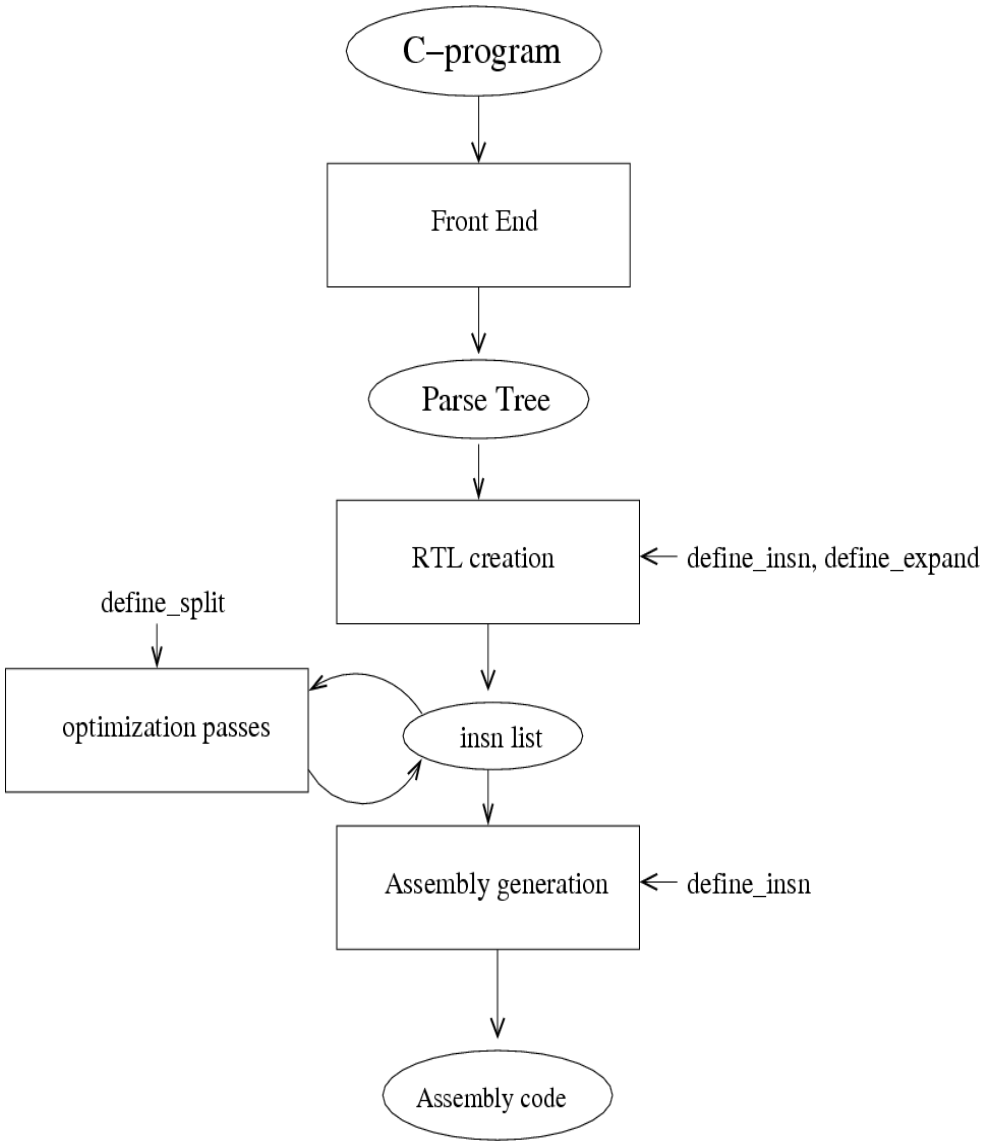

Figure 2: Usage of different *insns* in the compilation

First the named patterns, i.e., *define_ insn "addsi3"* and *define_ expand "iorsi2"*, are matched against the standard names [7] in the internal parse tree to create the RTL representation of the C language program. All the patterns can be named for commenting purposes, but names conforming to the standard names list have a special purpose in the RTL creation phase. The names intended for commenting

must start with an asterisk(*), so that they are not mixed with the standard pattern names. After the RTL creation the compiler optimizes the resulting *insn* list with a set of internal optimization routines and also uses the possible *define_split* definitions.

After all these phases the RTL list is converted into assembler instructions. In this phase only *define_insn* type definitions are used. The created RTL list is matched against all the *define_insn* patterns and the assembly code is output according to the output templates. Output assembly code can be described with regular assembly code strings or with C code for more complex ones. All the *define_insn* patterns are used in this phase whether they have a name corresponding to the standard names or not. In other words the compiler looks for similar *insn* patterns from the created RTL and the .md file, and executes the C code used to output assembly instructions or just outputs the assembly strings found in the template.

5.2.1 Example of define_insn

This type is used in the RTL generation phase if the name is part of the standard names list. All of the *define_insn* type patterns are used to generate assembly code whether it has a name or not. If the pattern is used only in the assembly generation phase and is named for commenting purposes, the name has to start with an asterisk. As an example below is the definition of the single integer addition of the COFFEE compiler port:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "register_operand" "=r,r,r")
(plus:SI (match_operand:SI 1 "register_operand" "r,r,r")
(match_operand:SI 2 "nonmemory_operand" "r,I,M")))]
""
"@
add\\t%0,%1,%2
addi\\t%0,%1,%2
addiu\\t%0,%1,%2"
[(set_attr "type" "arith,arith,arith")
 (set_attr "cc" "unchanged,unchanged,set")
 (set_attr "slottable" "yes,yes,yes")
])
```

First is specified the name of the pattern (**addsi3**) which tells us that this operation defines how to make an addition of single integers (SI) with three operands on the target machine (the number at the end of the name). Other possible types are: QI (quarter integer), HI (half integer), DI (double integer), SF (single float),

DF (double float). Also other types are possible but these are enough for almost all applications. When the name is part of the standard pattern names list [7] it is used in the RTL list creation phase and in assembly generation phase, as it is considered nameless at that phase.

Next is the RTL code defined, which the compiler adds to the RTL list whenever the compiler needs to do this operation, starting with the opening bracket. The keyword *set* defines that this operand receives the result of the *plus* operation and the *plus* operation itself has two operands.

The *register_operand* keyword tells the compiler that this operand must reside in a register. If it is not then the compiler outputs other RTL instructions to load it inside a register before this operation is performed. Other possible operand types are for example *memory_operand* and *immediate operand*. All of them are documented in the GCC internals manual [8].

The following single letters are used to indicate the type of register, i.e., floating point or integer. Also other types can be defined depending on the processor, or in case of immediates different immediate value ranges. Standard letters that can be used are defined in the GCC internals manual [8]. If you need other types then you have to define them in the header file (i.e., coffee.h). The macro **REG_CLASS_FROM_LETTER** returns the proper register class as described in another macro **REG_CLASS_NAMES**.

After the RTL expression the place with the empty string "" is reserved for possible run time checks. Usually these are used to distinguish between different architectures of the same processor family.

The part starting with an @-sign is the actual assembler output. Different lines separate the three cases in the RTL expression. The %0, %1, %2 etc. are marks for the compiler to replace the numbered operand in this position of the instruction. A set of output attributes are available to modify the details of the output operand in the internals manual [15], and there is a way to define additonal ones if the readymade ones are not suitable.

The assembler output can also be described as regular C code, as is the case in many of the COFFEE compiler ports instructions. Using C code the output is written as assembler instruction strings with a regular return-statement.

In the end we have *set_attr* type of statements. These are used to give assembly instructions attributes to define for example their type or their effects on the processors internal state. Their purpose and usage are explained in more detail later in this chapter.

5.2.2 Example of define_expand

This type of *define_* is only meant for the RTL creation phase. It takes no part in the assembly language output phase. As an example here is one *define_expand* definition from the COFFEE compiler port.

```
(define_expand "negsi2"
[(set (match_operand:SI 0 "register_operand" "=r")
(xor:SI (match_dup 0) (match_dup 0)))
 (set (match_dup 0)
(minus:SI (match_dup 0)
                  (match_operand:SI 1 "register_operand" "r")))]
""
"")
```

All expander definitions have to have a name corresponding to the standard pattern name [7] list. This has the name negsi2, which means that this *insn* handles the RTL-generation for a negation instruction for single integers (32-bit in COFFEE). As there is no negation for integers in the COFFEE instruction set this has to be handled by two different sequential *insns*. In this case two expressions are output: one to zero the target register (with a xor) and then subtracting the source operand from zero. It has to be noted that all expressions a *define_expand* outputs (both set expressions in this case) has to be matched by some *define_insn* pattern. In the COFFEE compiler port they are the standard pattern names *define_insn xorsi3* and *define_insn subsi3* respectively.

The *match_dup* expression means that this operand is the same as operand 0. Every operand can only be described by one *match_operand* statement and further references to this same operand has to be done with the *match_dup* statement.

5.2.3 Example of define_split

The Purpose of these definitions is to split complex *insns* into several simpler ones. Sometimes these complex *insns* require more than one machine instruction to be output. These cannot then be used to fill possible delay slots, so it makes sense to split them so that the scheduler can use the instructions resulting from the splitting in the delay slots.

Here is an example of one *define_split* definition:

```
(define_split
[(set (match_operand:SI 0 "gen_reg_operand" "")
      (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
 ""
[(set (match_dup 0)
      (ashift:SI (match_dup 1)
                 (const_int 16)))
 (set (match_dup 0)
      (ashiftrt:SI (match_dup 0)
                   (const_int 16)))]
"
{ operands[1] = gen_lowpart (SImode, operands[1]); }
")
```

Differently from other definitions, *define_split*s cannot be named. These definitions are used at the end of the compilation phase before assembly code is generated. The compiler uses these whenever it encounters an *insn* that is not matched by any *define_insn* pattern. At the moment the COFFEE compiler port does not have any *define_split* definitions.

First there is the RTL expression which needs to be split. Then the sequence of RTL-expressions replacing the original is described. In this case a sign extension operand is converted to a series of arithmetic shifts. This example is from a another processor description. The *define_split* can also be used to optimize the instruction so these descriptions might have some use in the COFFEE compiler port in the future.

5.3 Instruction Attributes

As seen in the *define_insn* example we can also give special attributes to instructions, e.g. type, effects on condition code or their applicability in delayed branch slots.

Attributes are like enumeral types in regular C code. You give a name to a attribute and the values it can have in a list. For example here is the definition of the type-attribute in the COFFEE compiler port:

```
(define_attr "type" "load,store,arith,branch,unknown,cmp,fadd,fmul,
                     fdiv,fload,fstore,fconv,fsqrt,fcmp,fabs,fmov,fneg"
 (const_string "unknown"))
```

The first name in quotes is the name of the attribute, after that is the list of possible text values it can have. Last we have the default value if none is given in *set_attr* section of *insns*.

There is a special attribute to describe the delayed branch slots. It is defined with a special *define_delay* definition. The COFFEE RISC core has one delayed branch slot after every branch instruction and it is defined like this

```
(define_attr "slottable" "no,yes,has_slot" (const_string "no"))

(define_delay (eq_attr "slottable" "has_slot")
  [(eq_attr "slottable" "yes") (nil) (nil)])
```

The attribute **slottable**(whose definition is given first) is tested and if an *insn* has the value **has_slot** then this instruction has a delayed branch slot. There can be a maximum of three delayed branch slots and the section with the square brackets is used to describe which instructions are suitable to be placed in this slot. The COFFEE has only one slot and every instruction with the attribute **slottable** with an value of **yes** can be placed in the first delayed branch slot. If no suitable instructions are found the compiler automatically outputs a nop-instruction into the delayed branch slot.

5.4 The Handling of Function Calls

In chapter four we introduced the ABI for the COFFEE RISC core. The conventions regarding the function calling sequence in the ABI need to be implemented in the COFFEE compiler port also. The implementation of the function calling sequence is spread between the files ”coffee.c” and ”coffee.h”, and it is handled by a few macros and functions shown here.

5.4.1 Register Allocation

The parts that define the register allocation in the COFFEE compiler port are handled by the macros in the ”coffee.h” file. The arguments that are passed by registers are defined like this:

```
#define FIRST_ARG_REG 0
#define MAX_ARGS_IN_REGS 5
```

These macros define the first argument register and the number of sequential registers available to pass function parameters. They can be defined very freely, as long as they are real physical registers of the processor. The number of argument registers, however, reduces the amount of registers available for variables and other

temporary values, so it is an important decision to make and has a considerable effect on the performance of compiled code.

The function return value is placed in the first argument register or the stack, if it is a structure or a union. The following macro

```
#define FUNCTION_VALUE(VALTYPE, FUNC)  \
  gen_rtx (REG, TYPE_MODE (VALTYPE), FIRST_ARG_REG)
```

tells the compiler to use the first argument register as the place for the return values. If the return value is a stack, the following macros are defined to tell the compiler to pass them on the stack of the caller function and the address of this is placed in the first argument register as a invisible first argument.

```
#define RETURN_IN_MEMORY(TYPE) (TYPE_MODE (TYPE) == BLKmode)
#define STRUCT_VALUE 0
```

The registers that need to be saved across function calls are handled by the macro

```
#define CALL_USED_REGISTERS { \
 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, \
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

and it is defined as an array with the size of the amount of physical registers the processor has, and if the value in the register is lost after a function call, then the value 1 is assigned to that registers slot in the array. The value 0 indicates the compiler to save that register in the stack during the function entry, and restoring it back before returning from the function.

The macro **CALL_USED_REGISTERS** can also be freely modified. The only things to consider are the argument registers and the stack pointer and frame pointer registers, which need to be defined as lost after function calls. Otherwise, it only affects the efficiency of the compiled code, and the best way to find the optimal number temporary registers and the call saved registers is by profiling some benchmark code with different number of call saved registers and temporary registers.

5.4.2 Stack Frame

The frame pointer and stack pointer are defined with the use of two macros in the ”coffee.h” file. They are

```
#define STACK_POINTER_REGNUM 27
#define FRAME_POINTER_REGNUM 28
```

which define the real physical register number reserved for them.

The function entry is handled in the "coffee.c" file and corresponding function is

```
void coffee_expand_prologue(void)
```

this function calculates the size of the stack frame needed for the current function under compilation, and allocates it by subtracting it from the value of the current stack pointer. Then the function saves the link register and the previous frame pointer to the stack, before updating the frame pointer to reflect the newly allocated frame. The last procedure of the function prologue is to the save the registers that need to be saved across function calls to the stack to their designated place.

The function that handles the exit is

```
void coffee_expand_epilogue(void)
```

and it recovers the saved registers and the frame pointer and the stack pointer, before returning from the function.

5.5 The GCC Low Level Runtime Library

Whenever the compiler encounters an operation, of which it does not know how to emit RTL code for, or there is no hardware support for it, it emits a library call to execute this operation. Typically these are arithmetic operations, e.g. integer division or floating point operation. Some of these operations are supported via machine independent C code, which is part of the GCC, but some of these operations have to be written in processor-specific machine code. This is only if your applications need these operations. Not supplying them does not hinder the compiler in any way if they are not needed.

All the possible library calls are documented in the GCC internals manual [17]. For the COFFEE compiler port the library calls to integer division(signed and unsigned) and the modulo operation are supported with machine coded libraries.

GCC can also emit library calls to certain C library routines, such as *memcpy* and *memset*. They are integrated into the GCC for the purpose of optimization. All the possible C library functions GCC might emit a call for are explained in the GCC user manual [18] (not the internals manual). These also have machine independent versions, but some have to be supported with processor specific machine code. Currently only *memcopy, bzero, bcopy* have COFFEE specific versions.

6. HANDLING OF SUB-WORD ACCESSES IN THE COFFEE COMPILER PORT

Since every processor has its own peculiarities, usually there are some obstacles in the porting process. The GCC has an ideal processor behind its porting architecture and differences between this ideal processor and the real processor create difficulties. In the COFFEE case one major difference is the lack of sub-word accesses. Sub-word accesses are loads and stores with values that are smaller than the internal register size of the processor.

6.1 Synthesized sub-word access

In the COFFEE port we had to emulate the half-word and byte accesses. The algorithm itself is quite simple but implementing it in the machine description was a tedious task. The pseudo algorithms for loading and storing a byte from a 32-bit word are described below. The same algorithm is easily converted to the half-word case.

Byte load algorithm

1. Mask the 2 least significant bits of the byte address
2. Load a 32-bit word from this aword aligned address
3. Multiply the 2 least significant bits of the original byte address by 8, denoted x
4. Shift the loaded word 24-x bits to the right (logically so that the byte is not sign extended)

Byte store algorithm

1. Mask the 2 least significant bits of the byte address
2. Load a 32-bit word from this aligned address
3. Multiply 2 least significant bits of the original byte address by 8, denoted x
4. Shift the byte to be stored 24-x bits to the left
5. Zero the bits, the byte starting from bit position 24-x, in the loaded word where the shifted source byte is to be stored.
6. OR the shifted source byte and the target word together
7. Store this resultant word back in the masked word aligned address

From these descriptions you can already see that not having a byte access in hardware results in significant performance penalty when handling bytes (or halfwords). Loading a byte is not a big problem but storing a byte is almost double in code size versus loading a byte. Depending on the application this might cause a memory bottleneck.

Similar algorithms as the ones described above are also used in the 16-bit accesses.

6.1.1 Implementation of sub-word Accesses

As sub-word access requires several assembler instructions, it has to be created with an expander definition. The COFFEE compiler port defines *define_expand "movqi"* and *define_expand "movhi"* for 8-bit and 16-bit accesses respectively. These patterns implement in RTL the same algorithm as described above.

The implementation of the sub-word accesses are handled in The C code part of a *define_expand* only. The C code section outputs the necessary *insns* in the RTL creation phase, and they are matched by regular *define_insn*s in the assembly creation phase.

There is, however, some additional code required to fully support the sub-word accesses. The compiler has a phase called reload where all the moves between the stack and registers are output. If a variable has no room to be in a register during the lifetime of a function, then it has a stack slot assigned to it. Every time when a variable is needed the compiler outputs the instructions to move this variable to a register or, if the variable has a new value written to it, outputs the instructions to move the value to the assigned stack slot.

In this reload phase the used patterns are called *define_expand "reload_inqi", define_expand "reload_outqi", define_expand "reload_inhi", define_expand "reload_outhi"* for byte load, byte store, halfword load and halfword store respectively.

The reason for these additional *define_expand* definitions is that the reload phase is handled after the RTL creation, and after that no new registers can be created for scratch values. In the RTL creation phase, every time a value or a variable is created it is assigned to a new register, even though there might not be enough physical registers in the processor. Later these extra registers are assigned to stack slots and further creation of additional registers is forbidden. So when a scratch register is needed later in the compilation phase, the compiler uses temporary register slots for these when it needs to move values in and out of the stack.

Below is the pattern for *reload_inqi* used in the COFFEE port:

```
(define_expand "reload_inqi"
  [(parallel [(match_operand:QI 0 "register_operand" "=r")
      (match_operand:QI 1 "picochip_reloadqi_memory_address" "r")
      (match_operand:DI 2 "register_operand" "=&r")])]
  ""
{
  rtx scratch, seq, addr;

  addr = XEXP(operands[1], 0);

  if (GET_CODE (operands[1]) != MEM)
    abort ();

    if (coffee_word_aligned_memory_reference(XEXP(operands[1], 0)))
    {
      /* Aligned reloads are easy, since they can use word-loads. */
      seq = gen_synthesised_loadqi_aligned(operands[0], operands[1]);
    }
  else
    {
      /* Get the scratch register.  Given an DI mode value, we have a
 choice of two DI mode scratch registers, so we can be sure that at
 least one of the scratch registers will be different to the output
 register, operand[0]. */

      if (REGNO (operands[0]) == REGNO (operands[2]))
scratch = gen_rtx_REG (SImode, REGNO (operands[2]) + 1);
      else
scratch = gen_rtx_REG (SImode, REGNO (operands[2]));

      /* Ensure that the scratch doesn't overlap either of the other
 two operands - however, the other two may overlap each
 other. */
      if (REGNO(scratch) == REGNO(operands[0]))
abort();
      if (REGNO(scratch) == REGNO(addr))
abort();

      /* Emit the instruction using a define_insn. */
      seq = gen_synthesised_loadqi_unaligned(operands[0], addr, scratch);

    }
  emit_insn (seq);

  DONE;
})
```

As the actual RTL *insns* are output only using C code the RTL instruction pattern (the three lines after the *define_expand*) consists of only the operands needed by this pattern. The DONE statement at the end of the C code section, as mentioned previously, indicates that this pattern should not output any more RTL *insns*, everything was handled with C code.

At first there is a sanity check to see that we have been given proper operands (the second operands is a memory reference). Then it checks whether the memory address is aligned or not (32-bit aligned), and depending on that it creates different *insn* sequences. At the end we have the function emit_insn(), which adds the created sequence to the *insn* list.

As the unaligned access requires scratch registers for address calculations and data manipulations, there is a macro in the C header file to inform the compiler about this situation. In the COFFEE port it is defined as follows:

```
enum reg_class
coffee_secondary_reload_class (enum reg_class class ATTRIBUTE_UNUSED,
 enum machine_mode mode,
 rtx x ATTRIBUTE_UNUSED,
 int in ATTRIBUTE_UNUSED)
{

  if ((mode == QImode) || (mode == HImode))
    return TWIN_REGS;

  return NO_REGS;

}
```

Every time the compiler needs to move a byte (QImode) or half word (HImode) between the stack and the register file, this tells the compiler to supply a scratch registers for the *reload_* patterns, otherwise no scratch register is needed and the compiler can use regular move *insns* in the reload phase also. The register class named TWIN_REGS is a special register class, which supplies two 32-bit registers for the *reload_* patterns because the scratch register might end up being the same as the destination register. This way we can be sure to have at least one register, which does not overlap with it. This definition is found in the coffee.c file, and it is used through the macro definitions in the coffee.h file. Below is their definition:

```
#define SECONDARY_INPUT_RELOAD_CLASS(CLASS,MODE,IN) \
  coffee_secondary_reload_class((CLASS), (MODE), (IN), 1)
```

```
#define SECONDARY_OUTPUT_RELOAD_CLASS(CLASS,MODE,OUT) \
  coffee_secondary_reload_class((CLASS), (MODE), (OUT), 0)
```

The same function is used in both cases because there is no difference whether we are loading or storing data between the stack and the register file.

6.2 Other Support Routines for sub-word Accesses

There are also two important C functions in the coffee.c file, which are used in the emulation of subword accesses:

```
void get_si_aligned_mem()
```

This function takes an unaligned memory address and converts it to a word aligned address and a bit offset from that address to the data (8 or 16-bit) in question. The bit offset is used to shift the data to the lower part of the register during load, and to move the data to the correct position for the store. These are then used in the *reload_* patterns to output the *insns* to handle the data transfer between the register and the stack.

The second function:

```
int coffee_word_aligned_memory_reference()
```

Checks if the memory reference is a word aligned reference to the stack. In this case loading and storing sub-word values is easier compared to unaligned cases.

6.2.1 Performance Impact of sub-word Accesses

The lack of dedicated instructions to access sub-word data on the COFFEE core causes a performance penalty, and depending on the application it can create severe problems to reach the required performance or timing (in for example real-time systems). It is advisable to consider carefully whether this can be a problem in your application, and if there is plenty of memory available, then it recommended to reduce the amount of sub-word data.

The current implementation uses 3-15 instructions per data access. The lowest amount is achieved when loading a byte from a word (32-bit) aligned address and the worst case is storing a byte to an unaligned address.

7. TESTING

The process of testing a compiler is very big task and represents a significant portion of the whole development process. It is not possible to guarantee that a compiler works 100%. The nature of compilers is that that they have unlimited amount of test cases (i.e., programs) and depending on optimization levels, many possible outcomes.

Compared to the amount of work that compiler testing requires it has quite little coverage in standard literature. For example the most known book about compilers (compilers:Principles, Techniques and Tools [20]) does not have any section about this process, but does bother to mention it briefly in a quote "Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

GCC includes a huge collection of many test cases, but these only test the features related to actual compilation process, i.e., whether it compiles standard compliant code without any errors or not. If the resultant assembly code is correct or not is not part of these tests. Running this test is automated and can be invoked from the build tree by running the make program with the command **make test**.

As the goal of this project was to develop a new processor with a completely new instruction set, there were a lot of problematic situations during the early development phase. As the development of the compiler started when there was only a HDL-description of the processor available (which was still under testing), the first tests of the compiler had to be done manually. This consisted of making small pieces of C code and then analyzing the assembly code by hand and trying to find incorrect code sequences.

At some point a larger program was compiled and analyzed with the aid of pen and paper. This is of course a very error prone and it is not a good method of testing a compiler, but given the circumstances we had very little choice. The program in question was an implementation of the DVD-decryption algorithm called deCSS [19]. This piece of software was very controversial when it was first released, but after 10 years from the release it is considered legal to use (at least for research purposes).

The reason for the use of this lawsuit prone software was the fact that it did not need any libraries to compile, it is mostly just mathematics and logical operations.

It is also reasonably small, so analyzing it by hand was not too tedious. It has not been actually run on the COFFEE core platform as doing that would require actual data from a DVD to be available in some way to our platform, but this has not been implemented and was not very high on the priority list.

To actually test the compiler in a more meaningful way requires a lot of other supporting software. As with all UNIX software GCC only does what is required, and relies on other software to do the rest. These jobs are handled by the assembler and the linker. Another important tool is the simulator, where you can run the code generated by these tools on virtual machine. The simulator executes a program on the processor and gives detailed information about the contents of the registers, the processors internal state, and how many cycles has the processor been executing. This way you can verify that your compiled programs are run correctly and provide the expected results.

7.1 Simulator Tests

Many different programs and pieces of code were run with the aid of the simulator. Most of them were not meaningful programs, but a few of them had some real world value and they were tested to evaluate the performance of the compiler. We run two widely used algorithms on the simulator, recorded the cycle count and compared the results against two other processors, which have a GCC based C compiler available. The processors in question are the Intel Pentium IV desktop processor and the current market leader in embedded processors, the ARM RISC processor.

7.1.1 FIR algorithm

A widely used algorithm in signal processing applications is *Finite Impulse Response* (FIR) Filter. It described in mathematical form as follows:

$$y[n] = \sum_{i=0}^{N-1} b_i x[n-i]$$

Where $y[n]$ is the output signal, $x[n]$ the input signal and b_i are the filter coefficients. N is the order of the filter, i.e., how many coefficients the filter has. FIR filter is basically just the weighted sum of N number of previous inputs, and if all the coefficients are set to the reciprocal of N, you get the average of N previous inputs. The main advantage of the FIR filter is that it is inherently stable, but it requires a lot of computational power.

The FIR filter was implemented with C, the code shown below:

```
int fir(void)
{
   int i,j,sum;

   j  = data_index - FIR_WEIGHTS;
   if(j < 0)
      j = j + MAX_DATA_SAMPLES;

   sum = 0;
   for(i=0; i<FIR_WEIGHTS; i++) {

      sum = sum + weights[i] * data_buffer[j];
      j = j + 1;
      if(j >= MAX_DATA_SAMPLES)
 j = 0;
   }
   return sum;
}
```

The resulting program was compiled and run on the previously mentioned processors and checked that the results were the same and the cycle counts were compared. The results are shown in table 4. These results are with values **FIR_WEIGHTS=8** and **MAX_DATA_SAMPLES=16**. On all the processors we used the same optimization level (-O1).

processor	cycles
COFFEE	137
ARM7	81
P4	86

Table 4: FIR filter cycle counts

The performance of COFFEE is clearly the worst. It is however reasonable considering that the whole COFFEE project has had only a handful of people working on it compared to the ARM and Intel processors and on the compilers. They have a much more complex instruction set, and in this case one advantage is in the addressing modes of instructions and the usage of conditional move instruction (which could be introduced to the COFFEE compiler port also). These instructions reduce especially the size of the main loop, which is the most critical part of this FIR algorithm. For example COFFEE needs 3 instructions to access each new *weights* and *data_buffer* value: one to shift left the loop variable to make it word aligned, one to add this value to the starting address of the array, and one to load the value from

the resulting address. Both ARM7 and P4 need only one instruction for this task and this gives them already a 24 instruction advantage in overall execution time.

7.1.2 Discrete Fourier Transform

Another algorithm run on the simulator to check the compiler correctness and evaluate the performance of COFFEE was the *Discrete Fourier Transform* (DFT). This algorithm transforms a signal from the time-domain to the frequency-domain, and it is a fundamental algorithm in signal processing used in, for example, data compression. In mathematical form it is written as:

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, k = 0, ..., N-1$$

Where:

- x(n) is an array of complex time-domain data
- n is an index of time steps
- X(k) is an array of complex frequency-domain data
- N is the size of the data arrays

The DFT algorithm was implemented in C, as shown below, and it was compiled with GCC on all processors with the same optimization levels(-O1).

```
void dft1()
{
  float pi2 = 2.0 * M_PI;
  float a,ca,sa;
  float invs = 1.0 / SIZE;
  for(unsigned int y = 0;y < SIZE;y++) {
    output_data[y].re = 0;
    output_data[y].im = 0;
    for(unsigned int x = 0;x < SIZE;x++) {
      a = pi2 * y * x * invs;
      ca = cos(a);
      sa = sin(a);
      output_data[y].re += input_data[x].re * ca - input_data[x].im * sa;
      output_data[y].im += input_data[x].re * sa + input_data[x].im * ca;
    }
  }
}
```

This implementation of the DFT is very heavy computationally, and it is not a recommended way of calculating it. It was used because of the ease of implementation, and meant only for testing the COFFEE compiler.

We run the resulting code on the COFFEE simulator and checked that the results were the same. Then we compared the cycle counts, which are shown on table 5. The **SIZE** was set to 10 in these examples. We did not have proper trigonometric functions implemented at that time so they were simply replaced with a constant value on all processors. This also removed the effect of the implementation of trigonometric functions from our examples as they can be very different on each processor/compiler.

processor	cycles
COFFEE	3971
ARM7	4598
P4	3350

Table 5: DFT cycle counts

This algorithm performed quite well on the COFFEE and its compiler. The performance is halfway between the ARM7 and P4. The ARM suffers compared to the COFFEE because it cannot load and store values from floating-point registers directly to the stack, and they need to pass through general purpose registers first.

P4 does not suffer from this and its powerful addressing modes provide the best results in this test.

As we were quite happy with the preliminary performance and correctness results, we moved next to actual hardware tests.

7.2 The COFFEE Platform

For the purpose of prototyping the COFFEE RISC core we developed an FPGA platform. This platform includes a lot of other hardware peripherals, which are needed to run programs and handle I/O. Also the COFFEE RISC core was modified to include the MILK coprocessor directly in the pipeline of the processor, and not as an external co-processor. This removed the overhead of moving data between the COFFEE and MILK processors. The resulting core was renamed as CAPPUCCINO.

The platform was realized on an ALTERA StratixII EPS2S180F FPGA device. Programming was done on a regular desktop PC, where we had our software development tools installed. The development tools consist of the COFFEE compiler port of GCC and the GNU binutils, which was also ported to the COFFEE RISC core and includes the assembler, linker, disassembler and many other utilities that are used to handle low-level object files. The source codes were cross-compiled and the resulting binaries were transferred to the platform using a regular serial port. The device also had an VGA port and that was used to display graphics and text on a monitor.

7.2.1 3D Graphics

As a part of his PhD research Fabio Garzia developed a set of algorithms in C for to be used in 3D graphics applications [26]. These algorithms were used for testing our platform and software development tools. After some debugging of the compiler and the hardware we could run a simple 3d graphics animation, consisting of a single rotating cube, on our platform.

The picture in figure 3 shows the FPGA prototyping board, and the monitor used to display the output of applications. In the figure 4 is a more detailed picture of the rotating cube.

The rendering of the cube was quite slow, but the main reason for it was the speed of the synthesized COFFEE RISC core on the FPGA device. It was run at an approx. speed of 50MHz, but an ASIC version could be about four times faster according to synthesis results [28].

Figure 3: The COFFEE platform and the 3D cube animation running on it

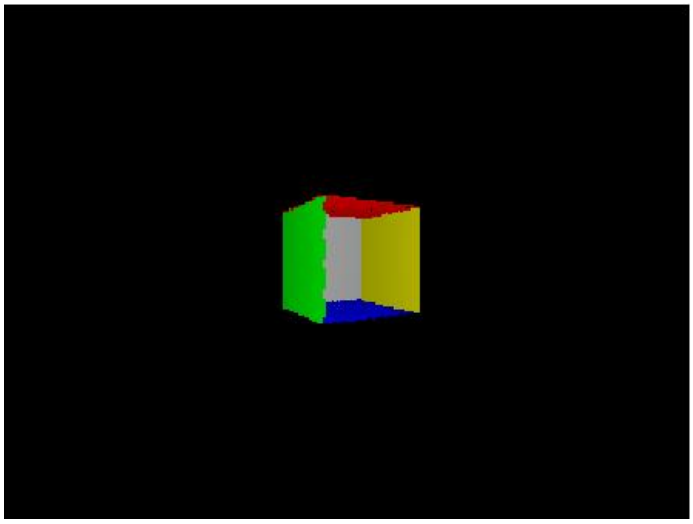

Figure 4: Close up of the rotating 3D cube

7.2.2 H.264 Codec

The H.264 [29] codec is a recent addition into the world of video codecs and it is used widely in different multimedia applications, such as the different internet streaming services and the Blue-Ray disc.

The baseline codec of the H.264 standard was implemented on the COFFEE platform and the focus was on the decoding part of the codec. It was coded in C and compiled using the COFFEE compiler port and the GNU binutils, and the resulting application was successfully used to decode a short video, which was displayed on the monitor connected to our platform.

7.2.3 Fast Fourier Transform

The COFFEE RISC core has been used in the development of a multi-processor SOC [25]. As a part of this research is the implementation of the *Fast Fourier Transform* [24], or FFT in short, for COFFEE platform. It is an algorithm used to calculate the DFT of a signal, but it is significantly faster than the one described earlier in this chapter.

The FFT was used to evaluate the performance of this algorithm on multi-processor platform that has 9 COFFEE RISC cores connected via a *Network-On-Chip* [27], also developed in Tampere University of Technology, but for comparison the results were generated for a single COFFEE RISC core using the COFFEE compiler port of GCC, and they are shown in table 6.

64-point FFT	cycles
radix-2	22,214
radix-4	10,937
radix-8	10,282

Table 6: Performance of 64-point FFT on COFFEE

7.3 Results

From the compiler development point of view the goal of all the applications and algorithms run on the COFFEE RISC core was to verify that the COFFE compiler port produced correct assembly code from C. After some debugging this goal was eventually reached.

The performance of the compiler was a secondary objective, as the GCC will produce decent code for all processors it has been ported to, because of the machine independent optimization algorithms. The performance of GCC is mostly affected

by the processors instruction set and the number of registers. Optimizing the back-end will not have much of an impact on the performance penalties introduced by a poor instruction set, or a limited number of registers.

The COFFEE RISC core has a good instruction set regarding its portability to GCC, and the results we have gained are quite good. The only negative point is the lack of sub-word accesses. The compiled C code according to the results is not severely lacking behind other, more optimized versions of GCC ports, and here lies the benefit of machine independent optimizations.

Some of the features of the COFFEE RISC core were also left out for the sake of simplifying the compiler port. One of them is the support for 16 bit encoding of instructions. Support for this would give a smaller memory footprint, but it also reduces the amount of registers and instruction available for the compiler to use, and this would result in a much slower performance. Especially the lack of registers in 16 bit mode would increase stack usage significantly, as there is not much room to keep variables inside the registers, or to pass parameters.

8. CONCLUSIONS

The purpose of this thesis was to develop a C compiler for a new RISC processor. Different ways of reaching this goal were analyzed, and based on this research we chose the open source compiler GCC, and decided to port it to our COFFEE RISC core.

The process consisted of developing an ABI for the COFFEE RISC core. Then a new back-end for GCC was developed, that followed this ABI and created assembly code for the COFFEE RISC CORE. The resulting compiler was tested by running C programs on a simulator, and later on real hardware.

Overall, the process has been a difficult one. There are many reasons to this, in particular the author's limited knowledge of compilers in the beginning of the project was a severe hindrance. The fact that the documentation of open source software is usually incomplete and not very user friendly, did not help either. The only way to really learn about the porting mechanism of GCC is to hack something together, compile code with it, and run it on a debugger and look at the different debugging outputs GCC provides when it encounters a problem.

The whole process of developing a compiler for a new processor architecture has been a good opportunity to gain detailed knowledge about compilers, different processor architectures, operating systems and everything that is required to put even a single character on screen in modern mobile systems, and they require a lot of supporting software and hardware, which one might never even have thought of, or the work it takes.

Despite all the troubles encountered, the compiler was developed successfully in the end and we used two different signal processing algorithms (he FIR and DFT algorithms) to evaluate its correctness and performance against other processors. The compiler produced correct code and we were decently satisfied with its performance compared to other processors.

8.1 Future Work

The work on the compiler is not over and there are several places for optimizations or new features. One future goal is to transfer the port to a 4.x version of GCC. This version has many more optimization routines than our current version, and it would give support for OpenMP [21] *Application Program Interface* (API). The OpenMP

is a parallel programming API, which is designed to be platform independent, and it can be used inside C/C++ code. The performance of the compiler can also be significantly improved by introducing, for example, the conditional move instructions, as was seen with the FIR-algorithm.

The whole COFFEE compiler port source code could also use some clarification, as it is filled with obsolete code resulting from the debugging process. Also the version of GCC source code was transferred during the course of this project from the 2.x to 3.x. The porting API of GCC has usually changed between major revisions, and this caused some additional work and the code became even more messier because of this.

Currently there is no support for the addition of debugging code in the code produced by the COFFEE compiler port. This would be a good addition to the whole COFFEE project, as the possibility of debugging the code run in the actual COFFEE platform would be much easier. Also, adding support for C++ code might be useful in the future, as it is gaining ground on embedded systems constantly.

As we have a complete GNU based toolchain we could also port the Linux kernel to our platform in the future. As the COFFEE does not have a *memory management unit* (MMU) our choice would be the ucLinux [23]. This kernel is designed for very small embedded processors who do not have an MMU, and the porting effort would be smaller than for a fully featured Linux kernel. On top of this we could then port the C Standard Libraries. There is a special version of the C libraries for embedded processors called ucLib, which is optimized to have a small memory footprint, but it does not have all the functions a full C library would have. It would, however, serve as good starting point and this would open up a lot of new possibilities for the whole COFFEE RISC project.

BIBLIOGRAPHY

[1] Juha Kylliäinen, Design and Implementation of Synthesizable RISC Processor Core, M.Sc Thesis, Tampere University of Technology, Department of Automation and Control Engineering 2004.

[2] C programming language, Wikipedia, the free encyplopedia, http://en.wikipedia.org/wiki/C-language.

[3] http://www.coware.com/lisatek.

[4] http://www.linux.org

[5] http://www.gnu.org/licenses/licenses.html#GPL

[6] Hirsjärvi, S., Remes, P., ja Sajavaara, P. 2005. Tutki ja kirjoita, 11. painos. Helsinki, Tammi. 436 s.

[7] GCC internals manual, section 16.9. http://gcc.gnu.org/onlinedocs/gccint

[8] GCC internals manual, section 16.8. http://gcc.gnu.org/onlinedocs/gccint

[9] Claudio Brunelli, M.Sc Thesis, Tampere University of Technology, Department of Information Technology 2003.

[10] Resve Saleh, Shahriar Mirabbasi, Guy Lemieux, Cristian Grecu, System-on-Cip: reuse and Integration. Proceedings of the IEEE, No. 6, June 2006

[11] GCC internals manual, section 17 http://gcc.gnu.org/onlinedocs/gccint

[12] The Development of the C Language, Dennis M. Ritchie, http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

[13] The C Programming Language, Brian Kernighan, Dennis M. Ritchie, Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.

[14] American National Standards Institute, American National Standard for Information Systems, Programming Language C, X3.159-1989.

[15] GCC internals manual, section 16.19 http://gcc.gnu.org/onlinedocs/gccint

[16] GCC internals manual, section 17.6 http://gcc.gnu.org/onlinedocs/gccint

[17] GCC internals manual, chapter 4 http://gcc.gnu.org/onlinedocs/gccint

[18] GCC user manual, section 6.51 http://gcc.gnu.org/onlinedocs/gcc/

[19] http://www.cs.cmu.edu/ dst/DeCSS/

[20] Aho, Sethi, Ullman, Compilers:Principles, Techniques and Tools, Addison-wesley 2006

[21] http://openmp.org

[22] http://developer.axis.com/wiki/doku.php

[23] http://www.uclinux.org

[24] http://en.wikipedia.org/wiki/Fast_Fourier_transform

[25] Roberto Airoldi, Fabio Garzia, Jari Nurmi, "FFT Algorithms Evaluation on a Homogeneous Multi-Processor System-on-Chip", Proceedings of 39th International Conference on Parallel Processing (ICCP) Workshops, to appear in September 2010.

[26] Fabio Garzia, Claudio Brunelli, Juha Kylliäinen, Markus Moisio, Jari Nurmi, "Design of a C Library for the Implementation of 3D Graphics Applications on a SoC". Proceedings of the 2006 International Conference on IP Based SoC Design, December 2006, pp. 371-374

[27] Tapani Ahonen, Jari Nurmi, "Hierarchically Heterogeneous Network-On-Chip. Proceedings of the 2007 International Conference on Computer as a Tool, September 2007, pp. 2580-2586.

[28] Juha Kylliäinen, Jari Nurmi, Mika Kuulusa, "COFFEE-A Core For Free", Proceedings of the International Symposium on System-on-Chip, Tampere, Finland, November 2003.

[29] T. Wiegand, G. Sullivan, G. Bjontegaard, A. Luthra, "Overview of the H.264/AVC Video Coding Standard", Circuits and Systems for Video Technology, IEEE Transactions, vol.13, no. 7, pp560-576, July 2003.

APPENDIX 1: COFFEE RISC CORE INSTRUCTIONS

The list of COFFEE RISC core instructions used by the COFFEE compiler port.

Instruction	Description	Notes
add dr,sr1,sr2	dr = sr1 + sr2	
addi dr,sr1,imm	dr = sr1 + imm	imm is a 15-bit signed immediate constant
and dr,sr1,sr2	dr = sr1 **and** sr2	
andi dr,sr1,imm	dr = sr1 **and** imm	imm as in addi
bc cr,imm	branch to PC + imm if carry	
begt cr,imm	branch to PC + imm if equal or greater than	
belt cr,imm	branch to PC + imm if equal or less than	
beq cr,imm	branch to PC + imm if equal	
bgt cr,imm	branch to PC + imm if greater than	
blt cr,imm	branch to PC + imm if less than	
bne cr,imm	branch to PC + imm if not equal	
cmp cr,sr1,sr2	Comparison between sr1 and sr2, and set the flags in cr accordingly	
cmpi cr,sr1,imm	Comparison between sr1 and imm, and set the flags in cr accordingly	imm is a 16-bit signed immediate constant
conb dr,sr1,sr2	Combine lower bytes from sr1 and sr2 to dr	
conh dr,sr1,sr2	Combine lower half words from sr1 and sr2 to dr	

Instruction	Description	Notes
jal imm	Jump to PC + imm and store PC to link register	imm is 25-bit signed immediate
jalr sr1	Jump to address in sr1 and store PC to link register	
jmp imm	Jump to PC + imm	imm as in jal
jmpr sr1	Jump to address in sr1	
ld dr,sr1,imm	Load dr with value from address sr1 + imm	imm is 15-bit signed immediate
ldri dr,imm	Load dr with immediate value imm	Pseudo instruction
ldra dr,label	Load dr with the address of label	
mov dr,sr	Move value from sr to dr	
mulhi dr	Store the upper 32-bits from 32-bit multiplication in dr	Only use after a 32-bit multiplication instruction
muli dr,sr1,imm	dr = sr1 ∗ imm	imm is 15-bit signed immediate
muls dr,sr1,sr2	dr = sr1 ∗ sr2	
muls_16 dr,sr1,sr2	dr = sr1 ∗ sr2	Uses the lower 16-bits of sr1 and sr2
nop	No operation	
not dr,sr	dr = **not** sr	Logical not
or dr,sr1,sr2	dr = sr1 **or** sr2	Logical or
ori dr,sr1,imm	dr = sr1 **or** imm	imm is 15-bit unsigned immediate
sexti dr,sr,imm	Sign extend sr and store to dr	imm is 5-bit unsigned immediate, indicates the position of the sign bit
sll dr,sr1,sr2	dr = sr1 « sr2	sr2 contains the shift amount
slli dr,sr,imm	dr = sr « imm	imm is 5-bit unsigned immediate
sra dr,sr1,sr2	dr = sr1 » sr2	sr2 contains the shift amount
srli dr,sr,imm	dr = sr1 » imm	imm is 5-bit unsigned immediate

Instruction	Description	Notes
st sr2,sr1,imm	Store sr2 to address sr1 + imm	imm is 15-bit signed immediate
sub dr,sr1,sr2	dr = sr1 − sr2	
xor dr,sr1,sr2	dr = sr1 **xor** sr2	Logical xor
fadd dr,sr1,sr2	FP addition	
fsub dr,sr1,sr2	FP subtraction	
fdiv dr,sr1,sr2	FP division	
fmul dr,sr1,sr2	FP multiplication	
fsqrt dr,sr	FP square root	
fconv.s dr,sr	Convert 32-bit integer to single precision FP	
fconv.w dr,sr	Convert single precision FP to 32-bit integer	
fneg dr,sr	FP negation	
fabs dr,sr	FP absolute value	